A Basic CWE-121 Buffer Overflow Effectiveness Test Suite

Paul E. Black, Hsiao-Ming (Michael) Koo, and Thomas Irish
U.S. National Institute of Standards and Technology
paul.black@nist.gov michael.koo@nist.gov thomas.irish@nist.gov

Abstract

Phase 3 of MITRE's Common Weakness Enumeration (CWE) Compatibility and Effectiveness program allows a customer to understand how effective a software assurance tool is at finding weaknesses and what code complexities it handles. Phase 3 is based on suites of test programs, but gives no criteria about how many programs are needed, their nature, how effectiveness is defined, or other details. We recommend principles in selecting a test suite for CWE effectiveness, and present a basic effectiveness test suite in C for CWE-121 Stack-based Buffer Overflow. For transparency we also document our steps in developing such the test suite. Finally, we suggest future work including code complexities.

1. Introduction

Some software assurance tools analyze source code, byte code, or binaries and report weaknesses in software. Their effectiveness depends on how their trade-offs between precision, recall, discrimination, programmer or analyst support, and many other factors matches the user's requirements.

The Common Weakness Enumeration (CWE) [1] is an "encyclopedia" of over 600 types of software weaknesses. MITRE established a CWE Compatibility and Effectiveness program, in which the "major aspect of the CWE Effectiveness phase is: ... to provide a public collection of test results that will allow a prospective customer to understand which CWE identifiers your capability is effective in locating; and, to articulate what types of complexity in software your capability is most successful at dealing with when looking for CWE identifier labeled weaknesses." [2]

The Software Assurance Metrics And Tool Evaluation (SAMATE) team at the U.S. National Institute of Standards and Technology (NIST) volunteered to develop an effectiveness test suite in one language, C, for one CWE as a prototype. We would develop principles and criteria at the same time. We chose buffer overflow as the weakness: it is reasonably well defined, it still occurs often, and we have thousands of candidate test cases. We considered several CWEs, but settled on CWE-121 Stack-based Buffer Overflow. [3]

Section 2 summarizes our background work in collecting and preparing candidate cases, some experiments we ran, and the selection process. In Section 3 we propose principles and criteria to guide the development of such test suites and present our test suite for basic CWE effectiveness. Section 4 notes issues that must be addressed in a test suite for code complexity effectiveness. In it we also suggest work to show that the suite conforms to the informal notion of effectiveness.

2. Our Background Work

Although MITRE gave general goals of the CWE Compatibility and Effectiveness program, they gave no guidance on the size or nature of a test suite or even how to measure "effectiveness." Our approach was to collect many candidate test cases, run many tools on them, and develop criteria and principles informed by tool behavior and comparison between cases.

The development of the basic test suite consisted of three stages. In the first stage we collected and prepared candidate test cases and acquired and installed static analysis tools. The second stage was a few experiments consisting of running the tools on different configurations of the cases. Finally, we came up with theoretical arguments informed and supported by analysis of the results to select a basic test suite.

2.1. Stage 1: Collect Candidate Cases

One of our first goals was to assemble many candidate test cases for CWE-121 in one place. We drew from four main sources, detailed below, and 17 other test cases from the SAMATE Reference Dataset (SRD) [4]. This resulted in 7338 test cases in 9962 source files, including six auxiliary files.

The Juliet 1.0 test suite "is a collection of C/C++ and Java programs with known flaws ... [covering] 181

ISBN 978-85-7669-274-4.

different kinds of flaws" [5]. Each test case had flawed or "bad" code and one or more corresponding pieces of "good" code, that is, the same code without the flaw. For CWE-121, there were 2946 cases. To keep analysis simpler, we split every case into two cases: one with only the bad code and one with only the good code. We also removed some unreachable code and conditional compilation (#ifdef) commands. This yielded 5892 cases in 8516 files.

Kendra Kratkiewicz developed 1140 cases for her dissertation [6]. Each test case had four variations: one buffer access which was within bounds, one access just outside the buffer, one a few dozen bytes outside the buffer, and finally one thousands of bytes outside.

From KDM Analytic's test case generator service we got about 17 000 cases with High, Medium, and Simple complexities and 0 to 5 call levels. We selected all the Simple complexities, a sampling from each High and Medium complexity and call level, giving 500 cases. Only 249 overflowed stack-based buffers.

We also included 41 benchmark programs which Fortify Software, Inc. donated in 2005 and 2006.

2.2. Other Sources of Candidate Cases

We believed these thousands of cases from three different sources was enough to begin. If these proved to be insufficient, there are many other sources.

The SRD has almost 600 buffer overflow cases classified under CWEs 120, 123, or 131. We did not use any of them since some may not be stack-based.

The SRD also has 28 moderate sized cases extracted [7] from applications, such as BIND and Sendmail. We did not use these since they were much bigger.

2.3. Stage 2: Run Tools on Test Cases

While collecting cases, we identified and installed a variety of tools. We chose flawfinder [8] and Cppcheck [9] as baseline tools. We also were granted research copies of several commercial tools.

Our next step was to run an experiment. Should tools be run on each case separately, or can tools be run on all cases combined into one program to save thousands of tool start ups? Handling large amounts of code is also more similar to production use. In theory there should be no difference: a weakness is found or not. In practice there are differences.

We adapted the approach taken for the Juliet test cases. To compile test cases separately or all together, we edited the test cases as needed, for instance, factoring out a `main()` function, adding conditional compilation statements around it, or removing it altogether.

The first production run for each tool used the compile-time option to treat all cases as one big program. We called this the Joint (J) run.

Second, we ran each tool against each test case separately. This should be the easiest for tools and should minimize any interference between different parts of the code. We called this the Individual (I) run.

Third, we made all typedef, function, and global variable names unique and concatenated all the separate files into one Massive (M) file of 28 Megabytes.

2.4. Experimental Notes and Observations

Only flawfinder produced exactly the same warnings on the Individual (I), Joint (J), and Massive (M) runs, because of minimal parsing; see Section 4.2. The differences between warnings produced by any tool were small. None of the runs were better predictors of a tool's performance on production software than others.

Run times varied significantly. Some tools ran I faster than J, and some ran J up to four times faster than I. One tool produced the same warnings for I and M taking two hours for I and only five minutes for M!

Our conclusion is that future work may use either Individual, Joint, or Massive runs, whichever is convenient, and get essentially the same result. This suggests that a single code complexity test case might contain weakness sites within several different complexities to reduce the amount of overhead and support.

We had two other observations. First, one tool could not handle duplicate function names, even in different name spaces. Second, some tools have notable size limits. Two tools could not handle J. We split J into up to a dozen subsets for these. One tool produced hundreds of warnings for M, but then ran out of memory.

2.5. Stage 3: Select Cases

With hundreds of thousands of results, we derived guidance for test cases. Since the structure of cases from each source was so different, we first examined each source separately, narrowing down each to a few of the basic or simplest bad cases. We chose groups of cases, decided if they were more basic than others, then articulated and codified objective measures corresponding to our subjective judgments. We repeated this until we had just a few cases from each source.

At one time we were left with some 15 test cases from all sources. We compared these, eliminating similar cases. In reviewing the remaining three, we decided two features were not included, so we selected two cases to cover those. Ultimately we did not need to write any new cases; existing cases satisfied our criteria.

3. Test Suite Principles

What principles should guide the development of a test suite for a different CWE or written in another language? How many test cases do we need? What is complex and should be left to the complexity test suite, and what is simple or basic? We propose the following fundamental principles for the whole test suite:

1. It is publicly available at no charge.
2. It consists of a small number of programs.
3. The programs are short.
4. They are based on reasonable criteria (theory) and on empirical evidence of utility (experiment).

Here are the justifications. A publicly available test suite may be debated and reviewed. A disadvantage is that a tool may have specific code to do well on the suite. We discuss this more in Section 4.1. By "small number," we mean five or ten—not hundreds. By "short," we mean no more than a page or two of code. A small number of short programs allows the suite to be understood through human review. Basing the test suite on theory supported by experience improves the chance that it truly measures effectiveness.

3.1. Principles for Test Cases in the Basic Suite

We propose the following principles. They eliminate the need for extra analysis steps and use common approaches. Some are specific to buffer overflows or the C language, and some are more widely applicable. Explanations follow each principles.

1. Use the simplest, common data type.
   For instance, `char` arrays instead of `struct` arrays. Use `int` instead of `long double`. Do not put data in containers, such as `struct` or `union`, which adds another layer of analysis.
2. Allocate data in a local scope.
   Global data has more potential code interactions.
3. Allocate data through declarations, not code.
   Using `alloca()` is uncommon.
4. Use constants or literals.
   The code to construct data takes more analysis.
5. Use the simplest, common library function.
   `strcpy()` for buffer overflow. For `strcat()`, existing content must be considered as well as buffer size. For `strncpy()`, the size-limiting parameter may prevent an overflow that otherwise might occur. `sprintf()` fills a buffer much like `strcpy()`, but its formatting makes determining the output length harder. `memmove()` is similar but uncommon.
6. Use minimal overflow or least violation.
   Writing one beyond the allocated buffer has slightly better discrimination than writing far beyond the buffer limit.
7. Eliminate unneeded variables, control structures, indirection, aliasing, or index computation.
   They are only needed to understand "what types of complexity in software" can be dealt with.

3.2. A Proposed Basic CWE-121 Test Suite

We propose SRD [4] Test Suite 81 as a basic CWE-121 Stack-based Buffer Overflow effectiveness test suite. It consists of the following five small programs. Each file name is followed by the SRD ID, the reason for the case, and the line(s) with the overflow.

- basic-00001-min.c 117
  very simplest; all tools but flawfinder report it
  `buf[10] = 'A';`
- basic-00034-min.c 249
  simplest case using pointer access
  `*(buf + 10) = 'A';`
- basic-00045-min.c 293
  simplest `strcpy()` case
  `strcpy(buf, "AAAAAAAAAA");`
- basic-00182-min.c 841
  simplest case with input
  `fgets(buf, 11, f);`
- stack_overflow_loop.c 1909
  variable index; bad bounds check: <= not <

```
for (unsigned i=1;i<=10;++i) {
    bStr[i] = (char)i + 'a';
```

4. Questions and Future Work

There is still work to be done in grounding the basic test suite better and developing a complexity test suite.

4.1. Grounding the Basic Test Suite

Research is needed to better understand the relation between tool behavior on a small test suite, behavior in production settings, and effect on software assurance.

Does order of functions matter? A massive file, detailed in Section 2.3, with cases in different orders can display if a tool has "warm up" or "fatigue" behavior, for instance, after producing several the same warning, a tool stops producing that warning. A tool may assume the first instance of a function call is correct, and flag subsequent instances as bad. A tool may treat functions P and Q differently if they are adjacent compared with when they are far apart.

A fixed test suite tends to create market pressure for tool makers to waste resources adding case-specific code or making rule changes. Creating obfuscated versions as needed may detect such short cuts, which will greatly reduce the pressure on tool makers: nobody wants to be caught "gaming" a test suite.

4.2. Developing a Complexity Test Suite

To complement the basic suite, we need a suite whose results "articulate what types of complexity in software your capability is most successful at" [2]. It is not feasible to develop a test suite that diagnoses every possible heuristic, trade-off, or design decision.

A guiding taxonomy, such as in [10], will help the test suite cover many code complexities. The suite might also have tests to display any limits of code size. Below are other facets to cover.

Consider code that can never be executed or that does not appear in the executable, for instance because of conditional compilation or lines commented out. Some users want warnings about unreachable code since after changes, it may be executed. Others only want to know about the actual behavior. The complexity suite needs cases to help users understand how a tool such these situations.

The following has a simple exploitable weakness.

```
(void)scanf(" %d", &j);
ar[j] = 7; // possible CWE-121
```

It is not exploitable if we add the guard `if (0<=j && j<=9) ar[j] = 7;` Should a tool report this as a weakness? The complexity test suite needs tests to help one understand how a tool treats exploitable or non-exploitable weaknesses.

Some tools do "fake" or minimal parsing [Chou, personal communication]. To determine this, the suite needs cases with small syntactic or semantic errors.

The complexity test suite needs cases *without* weaknesses to show that "the rate of false positives is less than 100 percent" as stated in CWE Effectiveness Tool Requirements A.2.10.

5. Conclusions

For large test suites tools may be run on test cases individually, all cases as one joined program, or combined into a massive file with little difference in result.

We believe that the five test programs we proposed in Section 3.2 constitute a useful suite in that running a tool on them yields "test results that will allow a prospective customer to understand which CWE identifiers [a tool] is effective in locating ..." We seek comments and suggestions.

Acknowledgments

We thank the following for help using their tools for this research: Red Lizard Software for Goanna, Monoidics for INFER, and Klocwork for Klocwork Insight™. We thank Gabriel Fan and Christopher Long for manipulating the thousands of test cases and running tools on them. We also thank Vadim Okun and Yan Wu for help acquiring, installing, and running tools.

Disclaimer

Certain trade names and company products are mentioned in the text or identified. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology (NIST), nor does it imply that the products are necessarily the best available for the purpose.

References

[1] "Common weakness enumeration," The MITRE Corporation. http://cwe.mitre.org/.

[2] "CWE compatibility and effectiveness program," The MITRE Corporation. http://cwe.mitre.org/compatible/program.html, Mar. 2010.

[3] "CWE-121: Stack-based Buffer Overflow," The MITRE Corporation. http://cwe.mitre.org/data/definitions/121.html.

[4] "SAMATE reference dataset," NIST. http://samate.nist.gov/SRD.

[5] T. Boland and P. E. Black, "Juliet 1.1 C/C++ and Java test suite," *IEEE Computer*, vol. 45, no. 10, pp. 88–90, Oct 2012.

[6] K. J. Kratkiewicz, "Evaluating static analysis tools for detecting buffer overflows in C code," Master's thesis, Harvard University, 2005.

[7] M. Zitser, R. P. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," in *Proc. 12th Internt'l Symp. on Foundations of Software Engineering*. ACM SIGSOFT, 2004, pp. 97–106.

[8] P. Broadwell and E. Ong, "A comparison of static analysis and fault injection techniques for developing robust system services," http://citeseerx.ist.psu.edu/viewdoc/summary?doi= 10.1.1.127.1508.

[9] D. J. Worth, C. Greenough, and L. S. Chin, "A survey of C and C++ software tools for computational science," STFC Rutherford Appleton Laboratory, RAL-TR-2009-028, Dec 2009.

[10] P. E. Black, M. Kass, M. Koo, and E. Fong, "Source code security analysis tool functional specification version 1.1," NIST, Special Publication 500-268 v1.1, February 2011.